AFRL-RI-RS-TR-2012-288

INTERACTIVE VULNERABILITY ANALYSIS ENHANCEMENT RESULTS

ASPECT SECURITY

DECEMBER 2012

FINAL TECHNICAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report was cleared for public release by the 88th ABW, Wright-Patterson AFB Public Affairs Office and is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2012-288 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR:

/ S /
FRANK H. BORN
Work Unit Manager

/ S /
WARREN H. DEBANY, JR.
Technical Advisor, Information
Exploitation & Operations Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
DECEMBER 2012	FINAL TECHNICAL REPORT	JUL 2012 – SEP 2012

4. TITLE AND SUBTITLE

INTERACTIVE VULNERABILITY ANALYSIS ENHANCEMENT RESULTS

5a. CONTRACT NUMBER
FA8750-12-C-0276

5b. GRANT NUMBER
N/A

5c. PROGRAM ELEMENT NUMBER
62788F

6. AUTHOR(S)

Jeff Williams and Arshan Dabirsiaghi

5d. PROJECT NUMBER
INTR

5e. TASK NUMBER
00

5f. WORK UNIT NUMBER
01

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Aspect Security
9175 Guilford Road, Suite 300
Columbia, MD 21046

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory/RIGA
525 Brooks Road
Rome NY 13441-4505

10. SPONSOR/MONITOR'S ACRONYM(S)
N/A

11. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFRL-RI-RS-TR-2012-288

12. DISTRIBUTION AVAILABILITY STATEMENT
Approved for Public Release; Distribution Unlimited. PA# 88ABW-2012-5735
Date Cleared: 31 Oct 2012

13. SUPPLEMENTARY NOTES

14. ABSTRACT
Interactive Application Security Testing (IAST) is an innovative new approach to application security vulnerability detection. This investigation focused on making performance improvements to this technology to allow deployments of the technology in production systems, to enable automatic generation of protection rules for the vulnerabilities discovered, and expanding the range of applications supported from JavaEE web based applications to other non-web based Java programs. Technology developed in this effort should be generally applicable to other IAST tools.

15. SUBJECT TERMS
Vulnerabilities, Malware, Application Security, Static Analysis

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			**FRANK H. BORN**
U	U	U	UU	38	19b. TELEPONE NUMBER (*Include area code)* **N/A**

TABLE OF CONTENTS

Section **Page**

LIST OF FIGURES

Figure **Page**

LIST OF TABLES

Figure **Page**

1. EXECUTIVE SUMMARY

Interactive Application Security Testing (IAST) is an innovative new approach to application security vulnerability detection being explored by Aspect Security researchers. Aspect Security was tasked by the Air Force Research lab to research strategic enhancements to this technology. The purpose of these enhancements was to improve the readiness of this technology for consumption by government agencies and the broader commercial market. Technical data in this report will cover the theory behind IAST technology and the results of the efforts to enhance this technology. It will not cover vulnerability testing itself. The tasks accomplished in this effort involved making performance improvements to allow production deployments, automatically generating protection rules from the vulnerabilities discovered, and expanding the range of applications supported. Technology developed in this effort should be generally applicable to all tools that use an IAST approach to vulnerability analysis.

Technical Approach

The three tasks required very different methodologies. The first task, to improve performance, required the use of profiling tools, application snapshots, line-by-line code review, and timing tools. The second task, to generate Web Application Firewalls (WAF) protection rules, involved code enhancements to the product, and thus required a lab machine with a vulnerable application on which rules could be tested. The third task, adding support for non-JavaEE languages involved building custom tools, using debuggers and profilers, and otherwise reverse engineering how the technologies targeted work.

Results

Aspect demonstrated that substantial improvements can be made to IAST, particularly in the area of performance. What follows is a brief summary of the results of the tasks. More details are available in sections 3, 4, and 5 below.

- **Task 1: Improved round-trip performance by 184x and startup time by 5x**
  Our test application had a round-trip time of 30ms without including IAST. Before our enhancements, the average round-trip-time (RTT) with IAST for the test application went was 950ms. We were able to improve performance to 35ms, only a 5ms increase over the baseline. Startup time was also reduced from almost 2.5 minutes to under 30 seconds. Running IAST in a production environment is now a possibility.

- **Task 2: Added Web Application Firewall (WAF) rule generation based on IAST results**
  The second task required research into how to correlate a IAST trace with a protection rule from the Enterprise Security Application Programming Interface (ESAPI) and ModSecurity web application firewalls (WAFs). We were able to automatically generate a WAF rule for 69% of the IAST rules. The other 31% were found to not be detectable or preventable by a WAF because of their nature. Generating a rule is a 2 click process that requires no input from the user.

- **Task 3: Added support for non-Java EE applications**
  Aspect's investigated a variety of Java-based technologies and how IAST can support them. We were successful in adding support for Scala, a popular new language, and determined that Cold Fusion support also looks possible. We developed several tools that make onboarding new languages and frameworks faster and more accurate. The methods developed during this task will allow IAST suppliers to make much more informed decisions regarding the product roadmap and product investment for a number of technologies.

2. INTRODUCTION

During the past decade, government dependence on application software, particularly web software, has grown immensely. As the complexity, interconnection, and criticality of the government's software applications grows, so too does the opportunity for malicious attackers to cause harm.

Finding and diagnosing vulnerabilities in an organization's application portfolio is an immensely complex task that requires large numbers of scarce qualified software assurance specialists. This situation cries out for automated tools that can help scale software assurance to the levels necessary. However, there are numerous challenges with existing automated tools for this purpose.

About Aspect Security

Aspect's team has focused exclusively on application security services since 1998. We have consistently demonstrated our leadership by contributing our research to the world through the Open Web Application Security Project (OWASP) industry group. Our methodology, tools, training, and services have evolved from our work securing millions of lines of code every month, teaching several hundred classes a year, and providing guidance to some of the most security-conscious customers in the world, including many different major financial, banking, e-commerce, and government customers.

Aspect researchers invented technology that can merge the power of static and dynamic analysis together and weave it directly into a running application. We use this interactive analysis to weave static and dynamic security tests directly into the application. From this new vantage point, we have the context to make security vulnerabilities easily visible.

Methodology

The following engineering staff was involved in delivering the product enhancements.

Task 1: Performance Improvements

- Arshan Dabirsiaghi
- Jeff Williams

Task 2: Automatic WAF Rule Generation

- Arshan Dabirsiaghi
- Alex Emsellem
- Matt Paisner

Task 3: Non-JavaEE Rules

- Jeff Williams
- Arshan Dabirsiaghi
- Stefan Edwards

3. METHODS ASSUMPTIONS AND PROCEDURES

This section provides an overview of the state of the art in application security automation and an introduction to the technology known as Interactive Application Security Testing (IAST).

Penetration Testing

For the past ten years, scanning, static analysis, and penetration testing have been the primary sources of software assurance. Web applications in particular have been dangerously easy to test and many of the flaws that have been discovered have been extremely damaging. The reason these application are so easy to test is their reliance on plaintext protocols and standardized data structures.

Most web penetration testing today assumes that the protocol and data structures are easy to intercept, parse, and modify. HTTP is all plaintext and so are the typical data formats sent by the browser, such as form data, xml payloads, and multipart requests. However, this has started to change in the past few years, and penetration testing is about to get significantly more difficult.

Another challenge to traditional penetration testing is the ongoing, rapid acceleration of software development cycles. The move to Agile brought cycle times down from 6 months to a few weeks. Now, DevOps is reducing those times to days, or even a matter of hours. The traditional assurance approaches simply can't operate in these timeframes. This is driving demand for new approaches for generating software assurance earlier in the lifecycle and more quickly.

The Future is Complex

The complexity of web application protocols and data structures is increasing quickly. First Ajax, sending requests too quickly to intercept manually, required filtering a flood of requests in real-time. Then, Web Services required complex XML documents and sophisticated tools in order to invoke a web method. Even more recently, application frameworks have moved to using serialized objects to communicate between code in the browser and web services on the server. Google Web Toolkit is a good example of the increasingly complex data structures being passed through this channel. Rich client and mobile applications using raw TCP sockets to communicate with servers are extremely difficult to test and require custom tools with custom data parsing. Many organizations are expanding their application architectures into mobile, HTML5, and WebSockets, which will lead to custom protocols and data structures that are difficult to test.

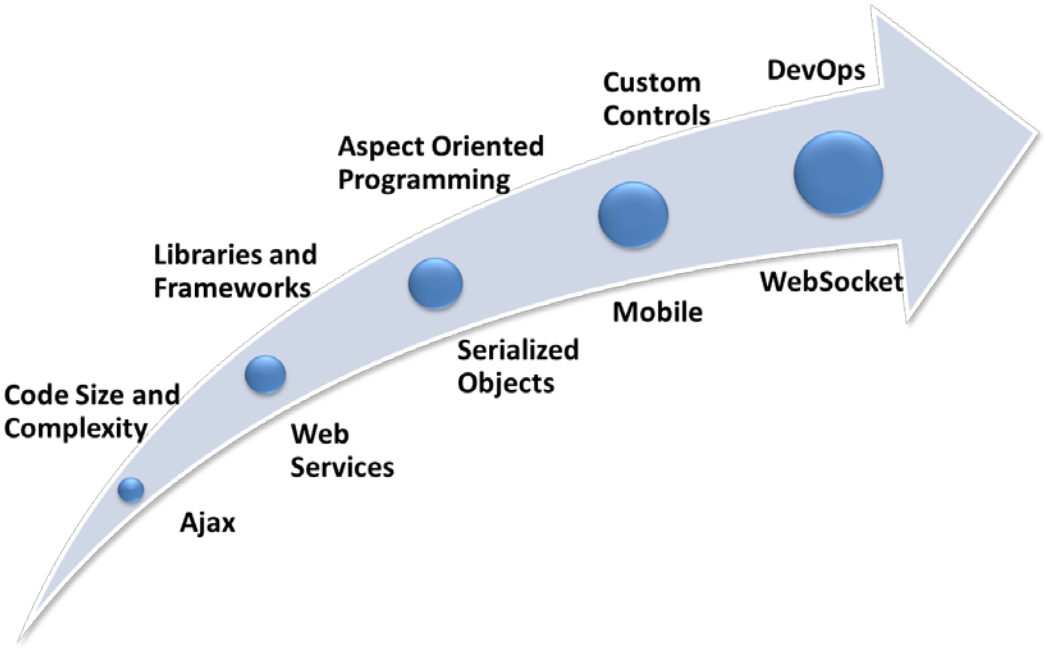

Figure 1. Factors Increasing the Complexity of Software Vulnerability Analysis

Organizations attempting to expand their security programs to cover their entire application portfolio cannot increase the level of effort to assure individual applications. Some organizations' portfolios have thousands of applications.

Using Automation for Software Assurance

Both Dynamic Application Security Testing (DAST) and Static Application Security Testing (SAST) can be helpful when used by experts as part of an application security program. But traditional tools don't stand much of a chance against the onslaught of new code, new frameworks, new protocols, and new data structures. Current scanning tools still don't do a very good job with Ajax and Web Services, technologies that are nearing the 10 year-old mark. And, static tools still struggle with false alarms and need to have new rules for every new framework and library that comes out.

Finding vulnerabilities in applications seems like it should be easy enough for tools to do. But the reality is that takes large amounts of ***context*** to detect vulnerabilities. To identify a vulnerability the tool has to understand how the code is invoked, what assets are involved, the permissions of the users, and many other details that are difficult to figure out automatically. Nevertheless, we desperately need automation to help us scale application security to the size it needs to be.

Interactive Application Security Testing (IAST)

Here's a short list of requirements for a viable application security tool:

- Simple enough for any developer to use, because most projects are not going to have a security expert available to run the tool.
- Must provide decent coverage over the entire application (including libraries) and be able to find all of the most common vulnerabilities.
- Must be extremely accurate, detecting flaws without false alarms that require time and expertise to diagnose.
- Must scale easily to the size of an organization's application portfolio.
- Must be affordable, as current tools are cost-prohibitive at portfolio-scale.

The idea behind IAST has existed for several years. The basic concept is that information from within the running application can help find vulnerabilities. IAST involves instrumenting the application with sensors that can monitor runtime activity, including control flow, data flow, software architecture, libraries, and more.

IAST is not the same as "hybrid" analysis. The idea marketed as hybrid analysis is that you could take the output from a dynamic scan and combine it with the output from a static analysis tool in order to get better output. In reality, these experiments have not been successful. Setting up and running both dynamic and static tools on an application and then triaging and interpreting the results, is time-consuming and requires expertise. And even then, combining the results doesn't yield much improvement in the signal-to-noise ratio.

Better Context Equals Better Results

The reason that IAST has the potential to do better than traditional techniques is that there is more contextual information available **inside** the application than *anywhere* else. From inside the application, IAST can see how the source code gets exercised in practice, how the data flows, and how the control flow operations actually work. Unlike dynamic tools, IAST can see into all of the code, including code that isn't exposed through external interfaces, like choosing a weak encryption algorithm. And unlike static tools, IAST has access to the entire runtime context, including the full HTTP request, values from files, database results, etc.

Using this contextual information, IAST tools can see vulnerabilities more clearly than tools that only have access to some of this information. Most people assume that a vulnerability is located in a single line of code in a particular file. But in reality, a vulnerability is a path that winds through a codebase from method to method, sometimes pausing in a data store before continuing to a point where damage occurs. Remember, IAST is not trying to detect an *attack*; rather, the goal is to proactively identify the path of that vulnerability through the code.

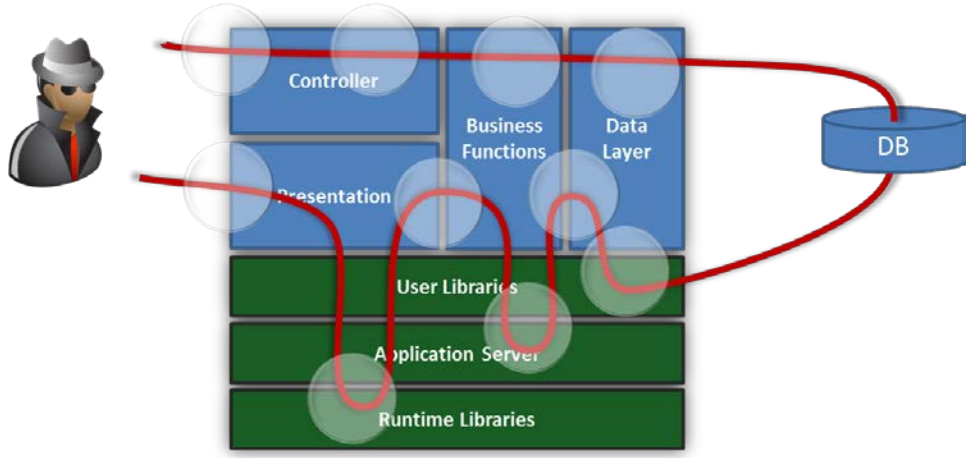

Figure 2. Software Vulnerabilities Are a "Path" Not a Single Line of Code

The figure above shows the trace of a stored XSS vulnerability path through a typical Java application. The data is extracted from the request in the controller, passed through several business functions into the data layer where it is stored in the database, then returned through the application to be included in the generated HTML by the presentation layer without encoding, completing the trace of a typical XSS vulnerability.

IAST can see all of the steps along this path, recognize the pattern, and capture a complete vulnerability description. IAST doesn't need expert testers because it has much more information than other approaches. And IAST doesn't require actual attacks, because it can see the vulnerable paths without actual attack data.

Assisting Traditional Tools

As you can see, IAST is a powerful technique for extracting useful information from an application. This information can be immensely helpful to traditional automated tools. For example, one problem shared by both dynamic and static tools is that they can't correlate URLs with the lines of code involved. This means that when a dynamic tool says that a particular URL is vulnerable to an attack, someone has to do a lot of work to figure out exactly where in a huge codebase that problem might be. Similarly, when a static analysis tool reports a problem on a particular line of code, someone has to figure out how to craft the proper series of web requests to test that the problem was actually fixed.

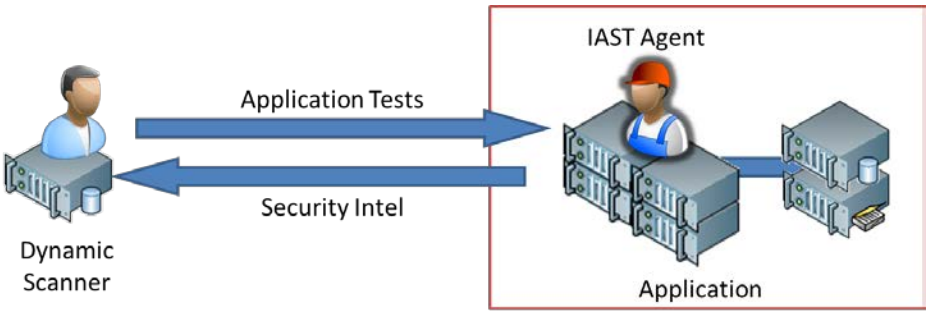

Figure 3. IAST Agents Provide Additional Contextual Information to Traditional Vulnerability Analysis Methods

As depicted in the figure above, the dynamic scanning tool does the vulnerability scanning, and the IAST agent sends back intelligence data about the attack surface of the application, whether attacks were successful or not, and a variety of other important information about the application.

Several vendors have produced IAST agents to assist with their traditional application security tools. HP has SecurityScope, which provides information back to WebInspect, and IBM has GlassBox, which supports AppScan Standard. IAST feedback could be used to improve static analysis results and is already helping dynamic scans produce more actionable, more accurate results.

Pure IAST

IAST does not require any external tools, as the analysis can all be done inside the IAST agent. This type of "pure" IAST agent doesn't need any external technology to do its work.

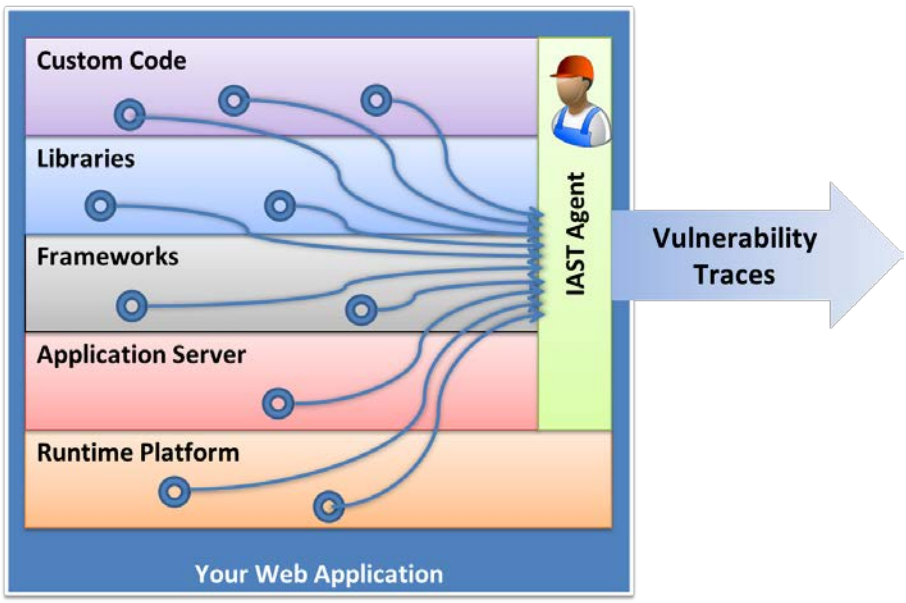

Figure 4. Pure IAST Agents Extract Context Across All Layers of an Application

In the illustration above, the pure IAST agent inserts passive sensors across the entire application stack, including all of the custom code and libraries. These sensors are installed using the instrumentation APIs available in most modern programming environments. Typically, these APIs are used to insert logging and performance calls, but they can be used to extract security information as well.

The sensors can extract a variety of information about the running application, enough to create a full picture of:

- Basic application intelligence, including lines of code, module count, and other metrics.
- Control and data flow, including the actual runtime values.
- Information about libraries and frameworks that are being used, and whether they are up-to-date.
- Architectural information about application structure and backend connections.

There are several advantages to the pure IAST approach:

- Can be installed simply, in just a matter of minutes, across a range of application servers.
- Vulnerabilities are reported immediately
- Can be run in a development, test, or production environment without affecting performance.
- In the development environment, the tool will provide insight into exactly the code that the developer tested, presumably what they are working on at that moment. This has the potential to provide developers with quick and accurate feedback right after mistakes are made, so they'll learn not to make those mistakes again.

Achieving Coverage

Probably the foremost challenge for automated application security tools is to ensure "coverage." Achieving coverage means that the entire codebase ("breadth") must be analyzed for a meaningful set of vulnerabilities ("depth"). While it might seem as though traditional, automated tools get good coverage, the numbers say otherwise. As you can see in the figure above, DAST tools typically only touch about 30% of the code in an application, less, if there is significant business logic that has to be navigated. The problem is that the crawlers just aren't smart enough to fill out forms and navigate wizards to exercise all of the functionality in an application.

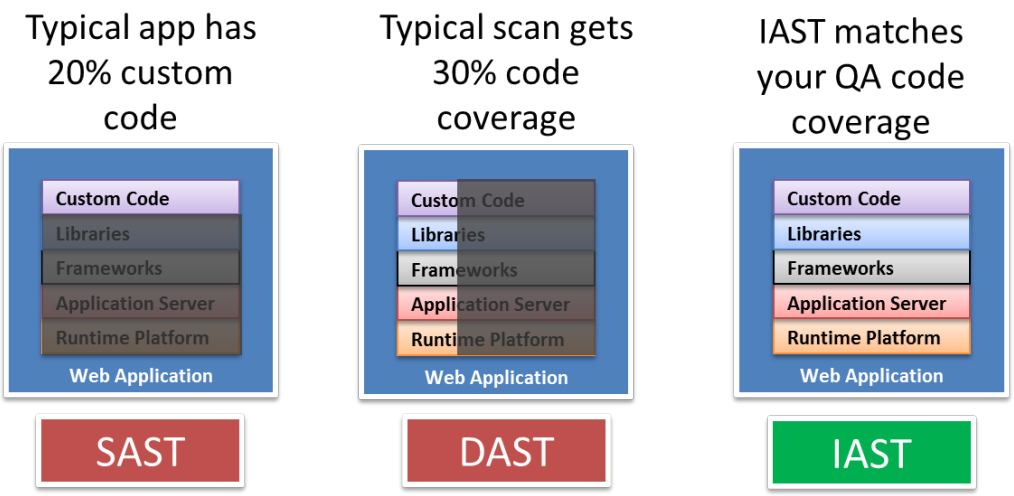

Figure 5. Comparing the Coverage of Static, Dynamic, and Interactive Application Security Testing

Also, while SAST tools can cover all of the custom code in an application, they don't look at the libraries and frameworks, which comprise over 80% of modern applications. The problem is that modeling millions of lines of library code would greatly expand the time and space requirements for a static scan. Instead, tool vendors model some of the libraries and simply mark the rest as lost sources or sinks.

IAST, on the other hand, covers exactly as much of the code as you cover in your automated and manual testing, including unit tests, system tests, and QA testing. While not usually 100%, organizations typically do fairly well with their test coverage. Since IAST watches any code that gets exercised, coverage usually exceeds that of traditional methods.

The Future

IAST may obviate the traditional penetration testing cycle. Instead of annual or biannual penetration tests on critical applications, organizations could move to a model where all of their applications are tested constantly, in the background, without the need for security experts. To get a complete security picture, perhaps some of IAST testing is done in development, some in QA, and some in production.

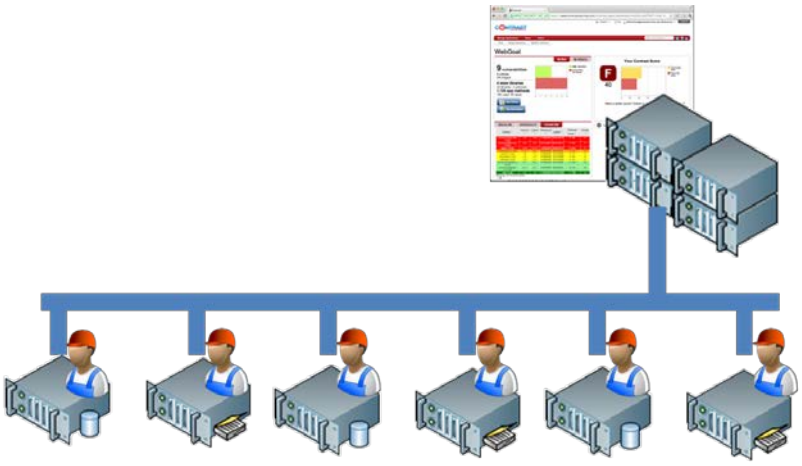

Figure 6 IAST Can Monitor an Entire Application Portfolio Continuously

Rather than receiving a one-time report, organizations could rely on having up-to-date security information at all times throughout the development process. And, this information could scale across their entire application portfolio. As you see in the figure above, all of the applications can be reporting their own vulnerabilities all of the time.

With better data, we can spend our precious, scarce, security expert time more wisely. Rather than having them chase yet another XSS or SQL injection hole, they could spend time designing new standard security defenses for the enterprise, that would make those vulnerabilities difficult or impossible to create in the future.

IAST is a natural match for newer development methodologies, such as Agile and DevOps. With releases occurring monthly, weekly, daily, or even many times an hour, traditional security processes simply cannot keep up. But IAST can instrument an application before the standard QA suite of test cases are run, perhaps right from the continuous integration server. As long as the test cases get good coverage, so does the security testing.

The future of IAST is bright. The potential is there to achieve much better vulnerability analysis results in a way that is more compatible with the way software is developed. As the software world moves to continuous integration, security needs to adapt and automate. The increased context provided by IAST tools can significantly improve the coverage, accuracy, and scalability of automated vulnerability analysis.

3. RESULTS AND DISCUSSION

Task 1: Performance Improvements

Aspect was tasked with investigating methods for increasing IAST performance for the purposes of making it usable in a production environment. The performance of the IAST engine will have a direct impact on its adoption and effectiveness. If the tool is slow to start, developers who need rapid test iterations will be unlikely to keep it enabled. If the tool slows down the application, it is unlikely to be used in a continuous fashion in production or near-production systems due to the effect on end-user experience.

Aspect was also tasked with identifying and prototyping additional or alternate methods of accomplishing vulnerability identification and reporting that have less effect on system performance. Aspect was also to prototype direct tracking of user controlled memory objects. This would eliminate the need for tracking tables which stored references to those objects, thus eliminating a relatively expensive hash table lookup on common operations.

To establish a baseline of performance, our JMeter test suite averaged around 6100ms on requests to Artifactory, an open source repository JavaEE application, with IAST running. Without IAST running, requests averaged around 100ms. Similar tests against WebGoat, a purposefully vulnerable JavaEE application, averaged around 950ms with IAST, and around 30ms without.

Optimizing Vulnerability Detection with Alternative Methods

Aspect was tasked with prototyping new and different ways of detecting vulnerabilities, in any step of the detection lifecycle.

Methodology

Our methodology was to profile the tool using various open source and commercial profilers, aggregate results, and micro-optimize hot spots and hope for noticeable performance gains.

Of the profilers tested, one was open source (jvisualvm) and three were commercial (JProbe, JProfiler and YourKit). Many difficulties were experienced trying to use the commercial tools to profile an application which had IAST running on it, due to conflicts between the simultaneous instrumentation performed by IAST and the profilers. Notably, the JProbe team worked diligently to provide a solution, but none of the features that distinguish the tool from open source solutions could be utilized for reasons that were never fully understood. The simplest and best performing tool was jvisualvm (also called VisualVM), which ships with Oracle's implementation of Java.

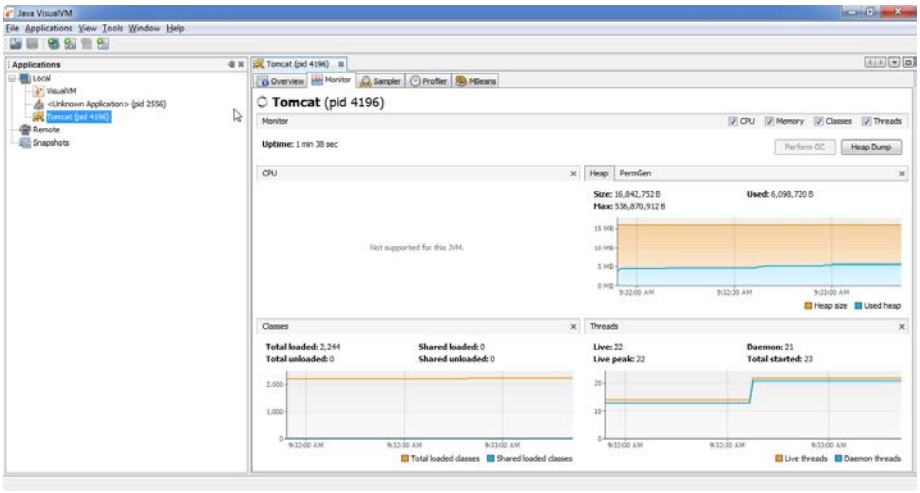

Figure 7. VisualVM providing simple CPU, memory, thread and class information

We took a large number of heap, Central Processing Unit (CPU) and thread snapshots during Java Virtual Machine (JVM) startup and during normal application usage to see if we could identify common execution bottlenecks. We also introduced metrics into the IAST system to measure how many times the tool's injected sensors were hit in the running JVM.

The following sections describe the details of the discoveries and enhancements made by profiling and analyzing sensor metrics.

Improvement 1: Extraneous Stacktrace Generation

It is important for IAST to capture stacktraces when notable security events occur because they contain code files and line numbers that will aid developers in fixing any associated vulnerability. However, stacktraces are extremely expensive to compute in terms of CPU, so they are only intended to be captured if an event is truly noteworthy.

Our first performance discovery was that for one faulty sensor, many of these stacktraces were being created before it was determined that an event was security relevant. This logic was moved after said test, and IAST then generated hundreds less stacktraces per HTTP request. It was difficult to quantify the impact of this change directly, because its CPU cost was usually consumed in other threads besides the one serving an HTTP request, and thus not easily measurable. We estimate we eliminated 600 stacktraces gathered per request, which on paper only reduces processing time a few dozen milliseconds. However, the rapid creation and release of these stacktraces caused garbage collector thrash that was difficult to measure directly but extremely costly.

After this change, the average request to Artifactory went from 6100ms to 500ms, an increase by factor of 12. However, even at 500ms, there was still a 500% increase on the 100ms average time without IAST. We believed this change removed most of the slowdown provided by the IAST sensors, and our metrics seemed to lend that assertion some credibility. If the sensors were no longer the bulk of our performance burden, we looked at the other operations of IAST for inefficiencies.

Improvement 2: Class Sorting and Short Circuiting

Next we created a test harness that would simulate the creation of events and keep track of the overall time. We discovered that the test harness was running almost 60ms faster than the actual IAST implementation. While we had been trying to improve the performance of the main IAST processing, we realized that perhaps that extra 60ms was being used by another less important thread. This led us to the IAST instrumentation thread.

Because classes in the JVM are lazy loaded, IAST must constantly keep tabs on which classes are newly loaded in order to install any needed sensors. This process turned out to be algorithmically inefficient, and eventually a quick, reliable way to short-circuit this entire process was discovered. The amount of Class objects loaded in a big application can be 30,000 or more, so having an unsound process around this process could result in many wasted cycles.

The way IAST identifies new Class objects began by polling the JVM every few seconds through the Java Instrumentation class's getAllLoadedClasses()[1] API. This method returns an unpredictably arranged array of Class objects that represent all the loaded classes on the JVM. IAST would convert this array to a List, and call removeAll() on that List with the HashSet of Class objects already seen in previous polls.

```
1: Class[] classes = inst.getAllLoadedClasses();
2: List newClasses = new ArrayList(Arrays.asList(classes));
3: newClasses.removeAll( checkedClasses );
4:
5: for(Class c : newClasses) {
6:  analyzeAndRedefine(c);
7: }
8:
9: checkedClasses.addAll(newClasses);
```

Figure 8. Original class update algorithm

Line 1 is a simple O(n) operation as the Class objects are loaded by the JVM into the resulting Array. Line 2 is 2 * O(n) as the the Array of Class objects is converted to a generic List subclass and copied to an ArrayList so that it can be manipulated. Line 3 is a O(n) as the List is searched linearly for every element of the given HashSet of Class objects already seen in previous polls. Line 5 is a simple O(n) as the reduced List of Class objects are analyzed and redefined if necessary. Line 9 is another O(n) as each item in the List of new Class objects is added to the HashSet of Class objects that represents those that have already been analyzed.

The total complexity for this entire operation is: $6 \times O(n)$.

[1] http://docs.oracle.com/javase/1.5.0/docs/api/java/lang/instrument/Instrumentation.html#getAllLoadedClasses()

It did not take long to come up with a major algorithmic improvement (and code simplification):

```
1: Class[] classes = inst.getAllLoadedClasses();
2:
3: for(Class c : newClasses) {
4:   if(checkedClasses.contains(c)) {
5:     continue;
6:   } else {
7:     analyzeAndRedefine(c);
8:     checkedClasses.add(c);
9:   }
10: }
```

Figure 9. Updated class update algorithm

Line 1 of our improved process is still O(n). Line 3 is an O(n) operation as well, with a larger constant time because it's searching over all of the loaded Class objects and not just the ones that hadn't been previously observed. Lines 4 and 8 are O(1) HashSet operations.

The total complexity for this entire operation is: $2 \times O(n)$. This is a substantial improvement overall the original complexity, $6 \times O(n)$. In the end, a few O(n) operations were traded for O(1) operations provided by HashSet.

However, the team realized that this constant barrage of checking for new Class objects might be unnecessary. We considered different approaches to short circuit the whole process:

1. Peel the top Class off the Array returned on Line 1, and see if it's the same as the last pass. If it was, that may indicate the value hasn't changed. This didn't work because the order of Class objects returned was unpredictable, and thus a change wouldn't necessarily indicate any change.
2. Create a quick hash of the Array returned on Line 1, and see if it's the same as the last pass. If it was, that may indicate the value hasn't changed. This didn't work for the same reason as 1.
3. Greatly increase the polling delay. We found that this helped performance, but would still cause "jerky" behavior whenever the polling occurred, and would hurt user experience as they would have to wait longer for newly loaded Class objects to have sensors installed. Thus their perception of how long it took for IAST to "start working" would lengthened.

Unfortunately, none of these approaches seemed feasible. However, we did stumble on a rather simple solution. When the Array of Class objects is returned by the JVM, we look at its size. If it's the same size, we return and assume no new Class objects have been loaded. This is not a guaranteed conclusion, because it's possible that X old classes have been unloaded and neatly replaced by X new classes. If this should happen, the changes will be picked up on the next cycle that does introduce a size change in the list. In our testing, this situation is unlikely and hasn't been observed to cause any detriment in any of our testing.

These two improvements greatly improve our "steady state" performance. There is practically no zero background thrash because after startup and initialization of an application, the polling operation does one O(1) operation and returns after it discovers now classes have been newly

loaded. After this operation, the average response to a WebGoat went from around 90ms with IAST to around 35ms, which was only 5ms slower than without IAST running.

Improvement 3: Unnecessary Propagation Sort

When IAST starts up, it installs sensors throughout the classes already loaded by default, which number around 1000. It was thought that this process was just very expensive, and there wouldn't be possibility for strong gains. Before any improvement work, startup took around 2m20s. There wasn't assumed to be any gains possible in this area, but profiling proved otherwise. Several snapshots taken during startup showed the process inside getPropagators() method.

A brief inspection of this code showed that it was sorting a List of Propagator objects on every call before returning. This method is called for each method in the JVM, which is at the very least several thousand times during the startup period. This sorting is completely unnecessary. The sorting functionality is only necessary for serializing policy files so that the Propagators show up in the resulting XML in alphabetical order. This was moved to another method, and was replaced with another method which returned an unaltered List of Propagator objects, which is all that is required by the processes at startup.

This simple change reduced startup time from 2m20s to 20-25s, an increase by a factor of almost 5.

Improvement 4: Message Compression

The profiler not surprisingly noted many CPU cycles being spent sending messages from the IAST engine to the tracking server application. Measuring those messages revealed they were much larger than we had been thinking. The average report was between 5KB and 45KB. Much of the I/O involved in communicating these messages to the server is thought to be a waste of cycles spent blocking. Thus, we increased the caching window on the client, lengthened the polling delay, and, perhaps most importantly added GZIP compression to all messages over 3KB in size.

Adding compression to messages trades CPU cycles for reduced time spent in I/O. Because I/O is always relatively slow, and our I/O layer is intentionally naive, we believed this was a good tradeoff. Thorough comparison between the performance of compressed and uncompressed messages should be done in the near future.

Optimization Conclusions

Aspect was extremely successful in reducing the performance impact of IAST on a running application. We did this by eliminating waste, performing macro-optimizations, and beginning to dabble in micro-optimizations. In our performance test suite, WebGoat takes about 30ms to respond to a given request without IAST. Before this effort, the same tests with IAST took about 950ms. At the end of the effort, the same tests took about 35ms, a 16% performance degradation. The team never predicted anything of this magnitude ever becoming possible.

Directly Tracking Objects in Memory

Currently, IAST marks an Object as "tracked" (or "tainted") by putting it into a ConcurrentWeakReferenceHashMap. When a sensor wants to know if an Object is tracked, it checks for the value in the given map. This is an O(1) operation, but with not-inconsiderable constant time repercussions. Aspect was tasked with establishing a way of adding state to Java Objects directly.

Methodology

The first thought may be to use the Instrumentation API to add a set of new member fields to the base Object class. However, the current Java Instrumentation API version explicitly disallows the adding or removing of any fields or methods during instrumentation. The documentation indicates that this restriction may be removed in future versions.

Our primary theory going forward was to use the "-Xbootclasspath/p" parameter to load an altered version of Object.class at the front of the primordial classloader. We experimented by adding a simple "tracked" boolean class member to Object.class. It is initialized to false, but could be set to true by IAST sensors.

The JVM failed to properly initialize with this change. This behavior has also been observed by others in the community[2], and seems to be rooted in the fact that the JVM assumes the size of an Object.class instance and fails to account for the change in the bootclasspath. However, given that most of the tracked variables in default usage of IAST are String objects, we tested adding this member to String.class and succeeded. The steps are relatively simple:

1. Get a copy of String.class from rt.jar (or classes.jar on Mac OSX)
2. Use a bytecode editor of bytecode transformer to add field to the class
3. Add the class to the "-Xbootclasspath/p" JVM switch before the target operation

Using this recipe, we could add members to the String, StringBuffer, StringBuilder, and other classes without disrupting the JVM.

However, after attempting to implement this change, we encountered several logistical issues:

- This would change IAST installation requirements and require them to not only set the -javaagent switch to point to the IAST engine, but also set the same value for -Xbootclasspath.
- Because the String class is different between Java versions, the user would have to correctly specify their Java version when downloading the Engine because an incompatible version provided by IAST would cause the JVM to fail.

[2] http://stackoverflow.com/questions/81786/can-i-add-new-methods-to-the-string-class-in-java

- Our tooling, all the way from our IDE, to our continuous integration server and through to our build machines would also have to be altered significantly to compile and test our new Engine that utilized these "hacked in" new members.

The feature would also not be future safe. The internals of the JVM are already specially tuned for the String class (see String interning[3]), and there's no guarantee that future optimizations of String wouldn't break the approach.

Results

Our conclusion was that while it would be possible to add state through the use of new fields in many Java Objects, the logistical complications that would be encountered with building, testing, maintaining and installing IAST would greatly outweigh the estimated 3% performance gain per HTTP request.

Direct Object Tracking Conclusions

Although adding state directly to Objects was possible but not logistically practical, the performance gains achieved in other areas were significant. Average round trip times to the test web application went from 950ms to 35ms when the round trip time without IAST was 30ms.

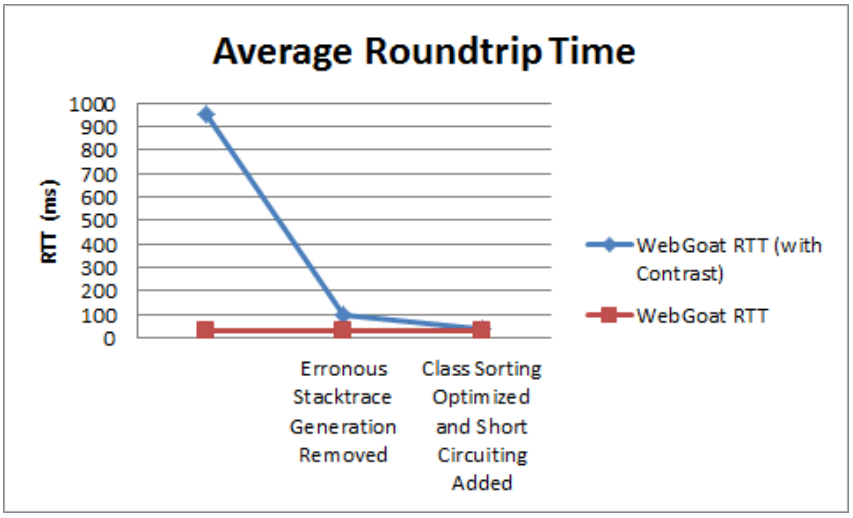

Figure 10. Average Roundup Time

[3] http://en.wikipedia.org/wiki/String_interning

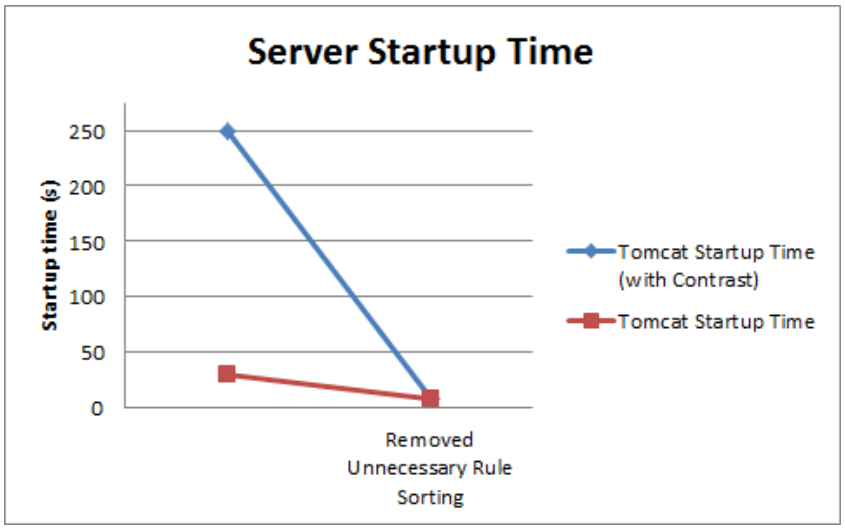

Figure 11. Server Startup Time

Task 2: Automatic WAF Rule Generation

Aspect was tasked with developing methods for automatically generating protection rules for use by software security wrappers (i.e., Web Application Firewalls (WAF)) to prevent attacks that target known vulnerabilities in the application under monitoring.

For this task, Aspect created the infrastructure, knowledge base and view for generating such rules and implemented rule generation for two popular WAF technologies: ModSecurity (http://www.modsecurity.org/) and the ESAPI WAF (http://www.esapi.org/). In the future, more WAFs could be added as requested by customers.

Methodology

In order to test the rules we created, we setup an Apache instance with ModSecurity as a reverse proxy in front of an Apache Tomcat with WebGoat. We designed a module within the IAST web application, which, when given a IAST trace, creates corresponding guidance information for the supported WAFs. Sometimes, security issues are not addressable from a WAF perspective, and in these cases the user can view the standard code remediation advice.

An example of such an issue is the usage of a broken cryptographic algorithm like MD-2. There's no reasonable way to alter the storage techniques of passwords from the WAF because it is fundamentally an application and database layer issue, not something that is visible in HTTP traffic that might be accessible to a WAF. Of the 19 rules currently projected to be ready for public use, 13 have corresponding WAF rule generation available.

Rule Design

The following table shows the rule design approach for several of the 13 rules for whose traces enforceable by a WAF.

Table 1. WAF Rule Approach for Several IAST Rules

Rule	ESAPI Rule	ModSecurity Rule
Injection Attacks from Parameters/Headers	A <virtual-patch> on the URI and vulnerable parameter with regular expressions specific to the vulnerability.	A SecRule scoped to the vulnerable URI that limits the canonicalized value of the input to the specified characters.
Lack of HttpOnly/Secure	An <add-http-only> or <add-secure-flag> rule.	A SecRule rule that detects cookie headers, and adds the "HTTPOnly" and "secure" flags if not found in the value.
Injection attacks from Cookies	A <bean-shell-script> that validates the individual cookie.	A SecRule that validates the Cookie header.

Generating a Rule

Users can easily generate these rules for themselves from within the prototype code. First, a user would select an individual Trace from the *View Traces* screen, hit *Generate WAF Rule* and select their targeted WAF (ESAPI or ModSecurity). The following figures show the generation process, including the generated rule.

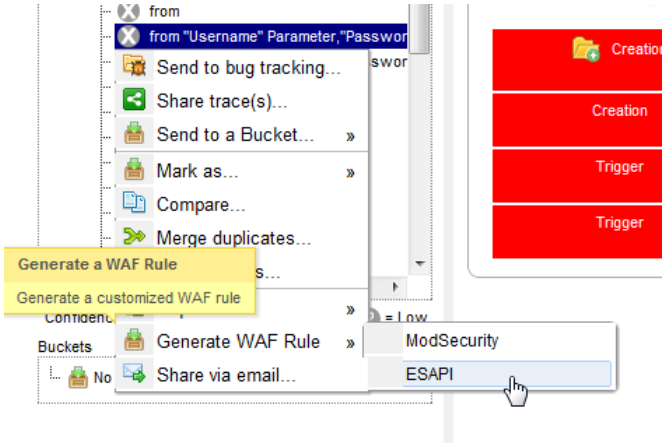

Figure 12. Rule Menu

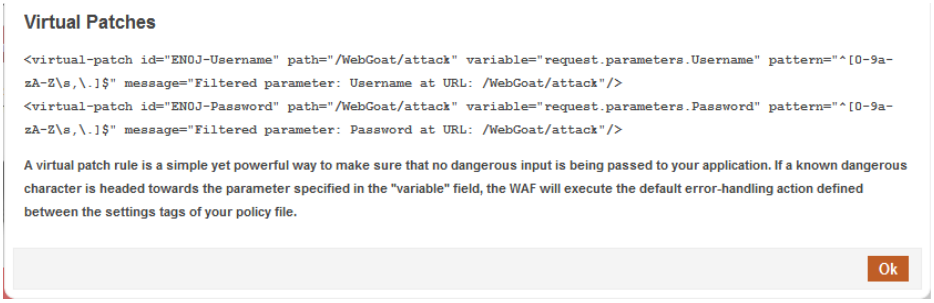

Figure 13. Virtual Patches

This particular trace had multiple sources of dangerous data, so fixing it requires multiple WAF rules, which are both shown to the user. The rules were all tested using WebGoat as a vulnerable application. To test ModSecurity, an Apache web server with ModSecurity was configured to front access to the Tomcat server on which it runs. To test the ESAPI WAF rules, the ESAPI library was added to WebGoat on a standalone Tomcat server.

Conclusions

The majority of flaws discoverable using IAST were found to have correlating WAF rules. Most of the rules were injection related, so they have a similar foundation. Some of the rules, like adding HTTPOnly to cookies, and preventing header injection, required more customization. Once the rules were created, it was a straightforward engineering task to implement these features into IAST.

Task 3: Non-JavaEE Rules

Aspect was tasked with extending the rule sets used by the vulnerability detection engine to analyze Java applications besides Java EE web applications. After research into several different areas besides classic Java EE applications, progress has been made in supporting some new languages, frameworks, and gained a clearer picture of what future investment would return in a wide range of technologies.

Methodology

To help identify potential sensors without a lengthy code review, we created an internal tracing tool called methodtracer. It's similar to the UNIX ltrace and strace tools, but simply echoes all the methods called in a Java process. To add these echoes, the tool uses a ClassFileTransformer to add simple println() calls to the start of method invocations, indented according to the stack depth. The purpose of this tool is to perform a function in the process (like navigate to a web page) and see all the methods that got called during that timeframe.

It takes a configuration file with a few simple options, like which packages to ignore during tracing, etc. The result is a simple internal tool for quickly narrowing down sensors in unknown JVM technologies.

With the help of methodtracer, the results of these forays into other languages is summarized by the following table:

Table 2. Feasibility of Performing IAST on Various Languages

Technology	Similar Technologies	Use Case	Market Outlook	Estimated Effort
Scala		Discover web application vulnerabilities in Scala.	Not large market, but trends towards early adopters and SaaS consumers.	Minimal. Initial tests show basic source-to-sink tests work in Scala without modification. Estimate 80 hours to test on multiple containers and add cases to test harness.
Play	None	Discover web application vulnerabilities in Play.	Not large market, but trends towards early adopters and SaaS customers.	Significant. The Play framework uses a lot of direct field access, instead of using accessor methods, which means IAST would have to insert field accessing sensors.

Groovy	JRuby, Jython	Discover web application vulnerabilities in Groovy.	Not large market, but trends towards early adopters and SaaS customers.	Medium. Research needed to discover sources, propagators and sinks. Also fatal issue with Java 7.
Applet	None	Act as an IDS or IPS against Java malware.	Value to AV vendors, and operational security teams. Too specific a concern for general population.	Significant. Green fields research into discovering generic exploitation conditions and methods. New reporting dashboards, new use cases, new market.
Desktop	Swing/AWT	Discover client-server vulnerabilities in J2SE applications.	Large. Enterprise customers review their whole portfolio, which would include desktop apps, and SMB vendors would value having an affordable security report to hand to customers.	Medium. Our research into creating rules to find meaningful vulnerabilities in desktop applications hasn't produced rules we think are commercial ready.

Detailed Analysis

The investigations and results of efforts to expand beyond traditional JAVAEE applications are detailed in the following sections.

Scala

Perhaps the most promising experiments were done with Scala. Running IAST on a Scala web application produced immediate, usable results without any customization or configuration. Scala happens to use the same APIs "under the hood" as your typical web application. Here is a screenshot of IAST finding a Cross-Site Scripting (XSS) vulnerability in a "Hello World" Scala application:

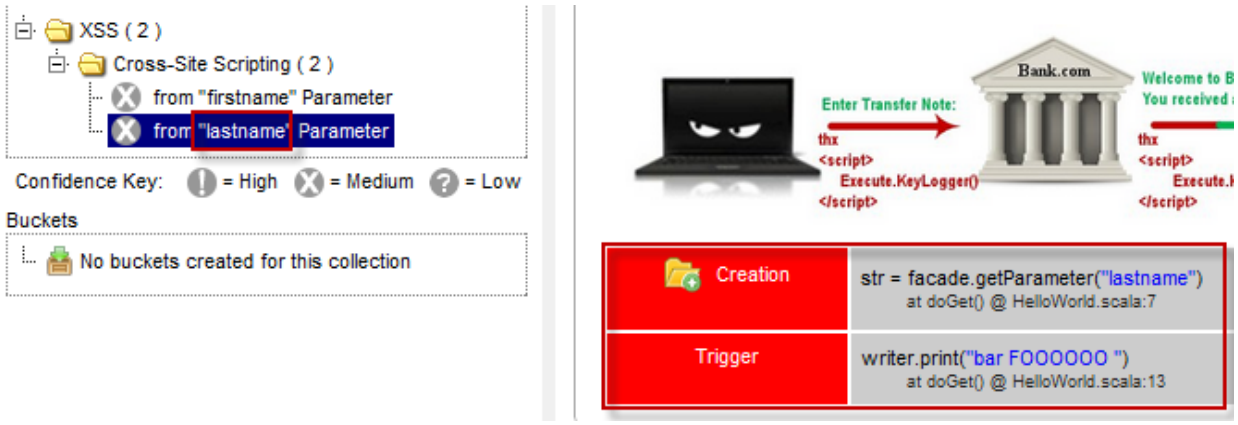

Figure 14. Invoking WAF Rule Generation from IAST Trace

Unexpectedly, even the correct filenames and line numbers are being resolved. Therefore, it appears that the sensors are likely adequate for detecting sources and sinks, but propagators should be researched and a set of Scala-specific test cases should be added to the test infrastructure before support can be officially claimed.

Play

Play was completely unknown to the researchers, so they used methodtracer to help identify sensors. While methodtracer worked effectively, it wasn't helping identify sources of untrusted data in Play. There were no methods observed which looked like sources.

```
10:00:28,740 INFO ~ Listening for HTTP on port 9000 (Waiting a first request to start) 10:02:20,040 INFO ~ Connected to jdbc:h2:mem:play;MODE=MYSQL
10:02:24,153 INFO ~ Application 'Yet Another Blog Engine' is now started !

...

play.server.StreamChunkAggregator.messageReceived(Lorg/jboss/netty/channel/ChannelHandlerContext;Lorg/jboss/netty/channel/MessageEvent;)V

play.server.PlayHandler.messageReceived(Lorg/jboss/netty/channel/ChannelHandlerContext;Lorg/jboss/netty/channel/MessageEvent;)V

play.server.PlayHandler.parseRequest(Lorg/jboss/netty/channel/ChannelHandlerContext;Lorg/jboss/netty/handler/codec/http/HttpRequest;Lorg/jboss/netty/channel/Messag
eEvent;)Lplay/mvc/Http$Request;

play.utils.HTTP.initLower2UppercaseHttpHeaders()Ljava/util/Map;

play.libs.IO.readLines(Ljava/io/InputStream;)Ljava/util/List;

play.utils.HTTP.parseContentType(Ljava/lang/String;)Lplay/utils/HTTP$ContentTypeWithEncoding;

play.utils.HTTP$ContentTypeWithEncoding.<init>(Ljava/lang/String;Ljava/lang/String;)V

play.server.PlayHandler.getRemoteIPAddress(Lorg/jboss/netty/channel/MessageEvent;)Ljava/lang/String;

play.server.PlayHandler.getHeaders(Lorg/jboss/netty/handler/codec/http/HttpRequest;)Ljava/util/Map;

...
```

Figure 15. Tracing Method Invocation During Execution

The trace above from methodtracer details the methods invoked when a request is received from a browser. The trace shows the exact method signatures that we can use to create rules in IAST. We use this trace output to assist in the identification of methods that could be used as sources, propagators, controls, and triggers.

We eventually discovered that Play makes untrusted data available to developer code by directly setting field members on instance level members of a class. IAST currently only supports tracking data via the use of methods, not fields. Therefore, tracking untrusted data through direct field access is something the product would require modification to detect.

To accomplish this, we would have to instrument all the *references* to these untrusted fields. Since these references might occur in any class, we would have to scan through all the instructions of every class and method to find them. This might not be a desirable feature for just supporting the Play framework given the performance hit during scanning each class this would require. There may be inexpensive ways of scoping that field level checking statically, but more research is required.

Given that there is no detection of sources without a lot of change to the IAST engine, further research seemed unnecessary.

Groovy

To test the usage of Groovy, Aspect used Ozone Widget Framework (OWF), an Open Source Software (OSS) widget development framework developed by NSA and published by DISA. At

its core, it's a web application written in Groovy. Testing immediately hit a roadblock when the Java process running OWF crashed when IAST was installed.

After spending days of effort digging into the crash (Oracle doesn't publish symbols so debugging is extremely difficult), a few things were learned:

- The crash occurs reliably on Java 1.7 on Sun/OpenJDK JVMs, but not on 1.6
- The crash occurs almost directly after IAST instruments a compiled Groovy script
- The crash occurs because an illegal class pool entry is detected

Here is a snippet from the crash file:

```
# A fatal error has been detected by the Java Runtime Environment:
#
# Internal Error (oops/constantPoolOop.hpp:372), pid=9112, tid=8096
# guarantee(tag_at(which).is_klass()) failed: Corrupted constant pool
#
# JRE version: 7.0_05-b05
```

Figure 16. Crash File Portion

Given that the crash occurs in Java 1.7 and not 1.6, we suspect it may have to do with Groovy's use of invokedynamic, a new bytecode instruction to be used by scripting languages on the JVM. Without the ability to debug the native JVM, it is not clear how to proceed forward.

Applet

Using IAST to detect when drive-by malware exploitation conditions occurred was discussed by the IAST team. There could be a use case where IAST would detect exploits like Trusted Method Chaining (like CVE-2010-0840) or Privileged Deserialization (like CVE 2010-0094).

This detection would obviously be useful in honeypot, IDS or IPS tools. This is a significant divergence from the current target market, deployment strategy, and use cases. It's possible the technology could be spun off or integrated into an existing antivirus or honeypot product. We briefed a member of the Oracle security team on the technology as it exists today and are hoping to follow up this thread to some useful conclusion, given the rampant exploitation of Java malware in real world attacks today.

Desktop

The team agrees that being able to run IAST on desktop applications would be beneficial, especially for customers who must assess the security of all their applications, including those that aren't web-based.

However, after some analysis, it appears the delivery of information from IAST should be markedly different from the web application, given the difference in threat model between web applications and desktop applications.

We project two common use cases for IAST with desktop apps:

- Use Case #1: As an infosec person or developer, confirm that an application developed by trustworthy individuals doesn't contain any vulnerabilities.
- Use Case #2: As an infosec person, confirm that an application isn't malicious.

Each of these cases is discussed in detail.

Use Case #1: Detect Vulnerability

After going through the existing rules, there was a very small subset that made sense to report for a desktop application:

- Use of an Insecure Encryption Algorithm
- Use of an Insecure Hash Algorithm
- Unchecked Read on Network Stream

Many of the other rules, like *XSS* and *Lack of HttpOnly* were specific to the web. Other rules could not be used to reliably locate vulnerabilities because insecure and intended functionality is indistinguishable without business context.

For instance, consider the SQL injection rule. IAST, out of the box, can be either set to trust or not trust data received from external entities (like servers, file system, networks, registry.). If the data received from those external entities is untrusted, there will be many false positives that result from the usage of some of those entities in a designed way. For example, a user may store their preferred sort order for a table in the UI of the application in a properties file. IAST would mark this source as untrusted, and a finding would result when it was used in a SQL query. The same would happen with data received from external servers if they were considered untrusted. A large number of false positives would result with an extremely rare true risk mixed amongst them and almost no way to quickly tell the difference. This low signal-to-noise ratio pattern we believe also justifies excluding most of the other existing rules.

Therefore, the remaining question is the addition of new rules to the rule set which specifically address desktop-specific risks. After spending several hours with various engineers, the team wasn't able to devise many generic rules for detecting desktop vulnerabilities. Most of the vulnerabilities Aspect's consulting practice discovers in desktop or mobile applications revolve around the notions of authentication, access control or other business-level concepts. IAST, just like other automated tools, is incapable, without training, of detecting issues with these abstractions. It can't decipher authenticated functionality from unauthenticated functionality.

Often in mobile application reviews, our team discovers that sensitive information is stored in an unsafe way. Either the data is unencrypted, encrypted with a key shared by all clients, or encrypted with an easily reversible password. IAST would have a very difficult time distinguishing actual vulnerabilities when looking for this type of issue. First, it wouldn't know which data is considered sensitive, so it would be forced to assume everything is sensitive. Next, it would have no way of knowing whether the key is shared amongst all users or specific to the given user or if it was easily reversible. IAST, with some modification, may be able to detect if a hardcoded constant is used as an encryption key, but there will be a high false negative rate. A

hardcoded constant key will likely be passed around from function to function before being used, causing it to appear as a function parameter instead of a constant.

Use Case #2: Detect Malice

A similar problem occurs when an information security person would use IAST to detect if an application was malicious. Malicious behavior is not easy to signature without having a large amount of false positives. There's no automatic way to tell if an application is phoning home state secrets or invoking a Web Service to look up harmless metadata. To IAST, they would look indistinguishable.

Another example of malicious behavior involves file operation. It's difficult to tell if an app is saving a configuration file or overwriting important application data for Microsoft Word. Running native processes may be a tip off that the program is malicious, but it's possible the app is just using a command line call to get some process information not obtainable within Java.

Hopefully it has been shown that it is difficult for the IAST engine to distinguish malicious behavior from non-malicious behavior. However, if the behavior is presented clearly enough to an infosec person, perhaps malicious intent can be noticed.

Abstracting these various behaviors, and others, into a screen which would allow a security specialist to easily understand an application's footprint in the context of its business person would streamline the detection of malicious applications. Here's a prototype view that we think could be useful in this use case at a high level. We intend the user to have a one more level of depth in "digging in" to the capabilities highlighted.

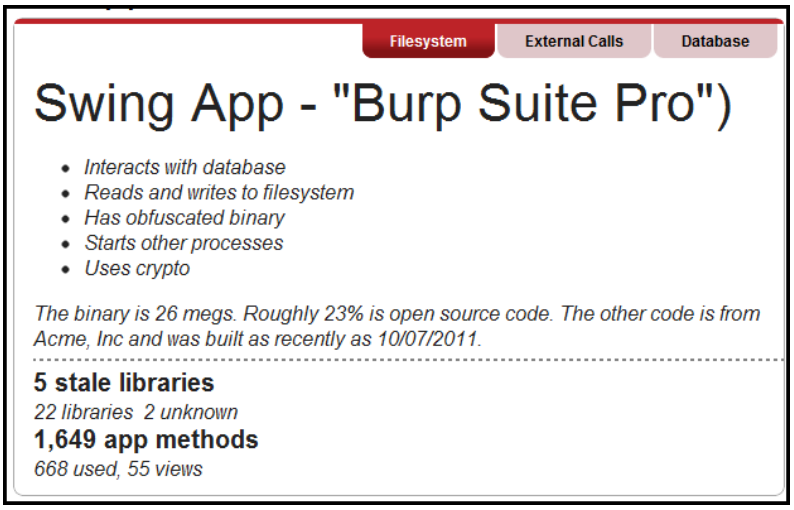

Figure 17. Prototype screen showing application security footprint based on IAST analysis

This screen is a movement away from the typical use case of finding vulnerabilities towards observing behaviors, which must be supported by a human with business context.

Conclusions

Some of the technologies, like Scala, may only require ruleset expansions, if anything at all, while some would require changes to the IAST engine, like Groovy and Play. We think applets and desktop applications would require a lot of work to satisfy their common use cases, both in the engine and site. This effort was extremely helpful in gauging the work necessary to reach full support in many of the technologies analyzed.

6. FURTHER RESEARCH

The results from this research were extremely positive, including illuminating areas in which IAST could continue to improve and expand to other areas of concern to DOD and the private sector.

1. **Enhanced MethodTracer** – Adding new platform support to an IAST engine requires some understanding of the exact paths that data and control flow follow. To more easily reverse-engineer this information from a new application enviornment, better tools are needed. We suggest research into quickly analyzing and capturing this information for new envronments, including the uxe of data canaries.

2. **Dynamic Remediation Instructions** – Can IAST generate customized remediation advice specific the application and frameworks, as opposed to generic advice. We propose to analyze the context of each vulnerability to determine all the technologies involved, the vulnerable code, and the security controls available. Using this context, we will automatically generate custom remediation advice in a form easily implementable to the developer. The goal is to provide a copy-and-paste fix for developers to use.

3. **Enhanced Coverage Measurement** – Measuring the accuracy of vulnerability tools has dominated tool evaluations, and resulted in tools that only search small portions of applications. Measuring coverage is at least as important. We propose to create coverage measurement and reporting tools that will help accurately benchmark tools.

4. **Automated Risk Profiling** – We propose using IAST to detect backend databases and other important services and data sources, using this map of data resources to identify, rate, or label, sensitive data, linking this data back to all applications that consume it, and calculating a portfolio risk rating for each application, prioritizing its importance.

7. ACRONYMS

CPU – Central Processing Unit

CVE – Common Vulnerability Enumeration

DAST – Dynamic Application Security Testing

ESAPI – Enterprise Security Application Programming Interface

IAST – Interactive Application Security Testing

IDS – Intrusion Detection System

IPS – Intrusion Prevention System

JVM – Java Virtual Machine

OSS – Open Source Software

OWASP – Open Web Application Security Project

OWF – Ozone Widget Framework

RTT – Round Trip Time

SAST – Static Application Security Testing

WAF – Web application Firewall

XSS – Cross-Site Scripting